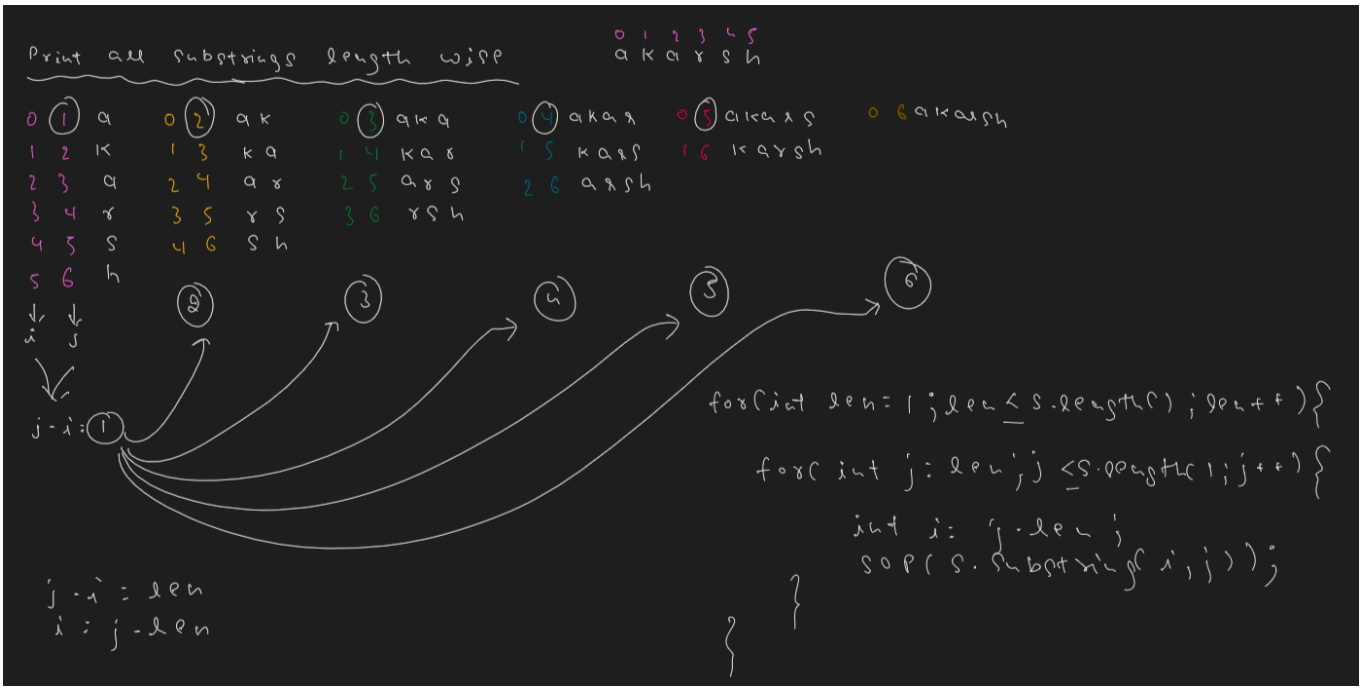
✅ Print All Substrings Length Wise




```java
public class Main {
	public static void main(String[] args) {
		String s = "akarsh";
		printLenWise(s);
	}

	public static void printLenWise(String s) {
		for (int len = 1; len <= s.length(); len++) {
			for (int j = len; j <= s.length(); j++) {
				int i = j - len;
				System.out.println(s.substring(i, j));
			}
		}
	}
}
```

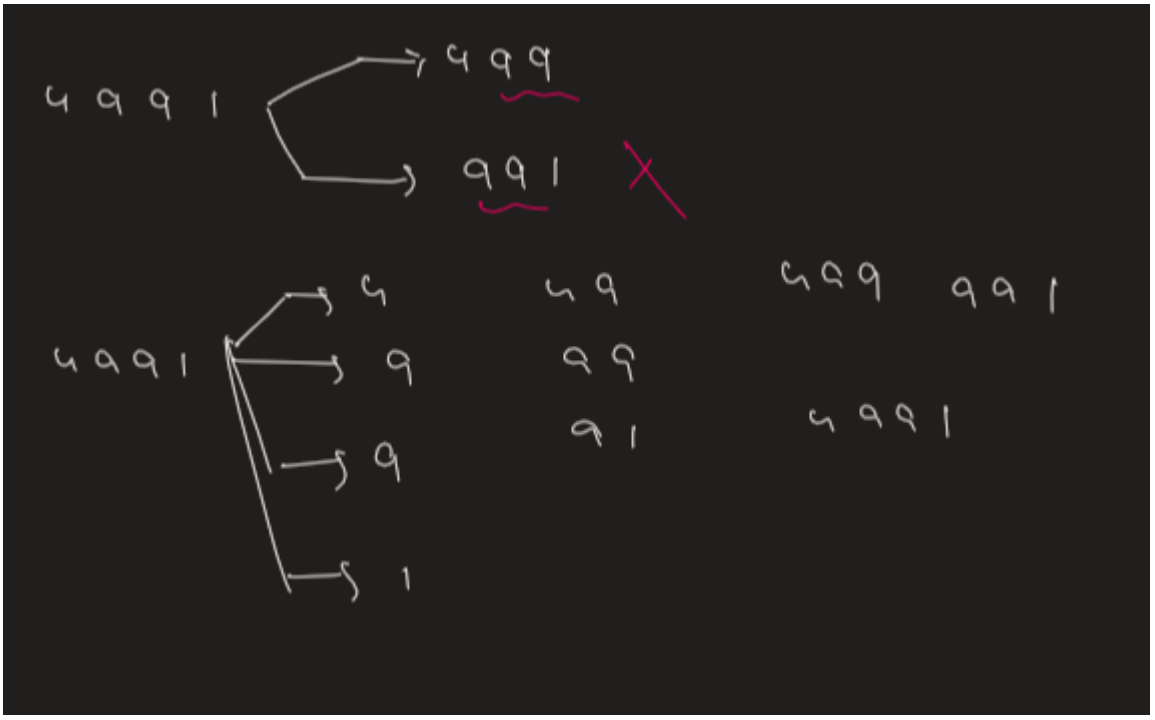
🚀 String to Primitive Data Type Conversion

```java
public class Main {
	public static void main(String[] args) {
		String str = "5467890";
		int x = Integer.parseInt(str);
		x++;
		System.out.println(x);

		String str1 = "54678965780";
		long a = Long.parseLong(str1);
		System.out.println(a);

		String str3 = "54678.901";
		double d= Double.parseDouble(str3);
		System.out.println(d);
```

```
    }
}
```

🎯 Finding CB Numbers

https://codeskiller.codingblocks.com/problems/165



```java
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        String s = sc.next();
        cbNumber(s);
    }

    public static void cbNumber(String s) {
        int c = 0;
        boolean[] visited = new boolean[s.length()]; // false

        for (int len = 1; len <= s.length(); len++) {
            for (int j = len; j <= s.length(); j++) {
                int i = j - len;
                long num = Long.parseLong(s.substring(i, j)); // i to j-1

                if (isCBnumber(num) == true && isvisited(visited, i, j) == true) {
                    for (int k = i; k < j; k++) { // mark it i to j-1
                        visited[k] = true;
                    }
                    c++;
                }
            }
        }
```

```
        System.out.println(c);
    }

    public static boolean isvisited(boolean[] visited, int i, int j) { // i to j-1
        for (int k = i; k < j; k++) {
            if (visited[k] == true) {
                return false;
            }
        }
        return true;
    }

    public static boolean isCBnumber(long num) {
        if (num == 0 || num == 1) {
            return false;
        }

        int[] arr = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == num) {
                return true;
            }
        }

        for (int i = 0; i < arr.length; i++) {
            if (num % arr[i] == 0) {
                return false;
            }
        }

        return true;
    }

}
```

🧱 Wrapper Classes

In **Java**, *wrapper classes* are used to **wrap (encapsulate) primitive data types** into objects. This allows primitive types (like `int`, `double`, etc.) to be used where only **objects** are allowed—such as in **collections**, **generics**, or **method arguments that expect objects**.

🔑 Why Wrapper Classes?

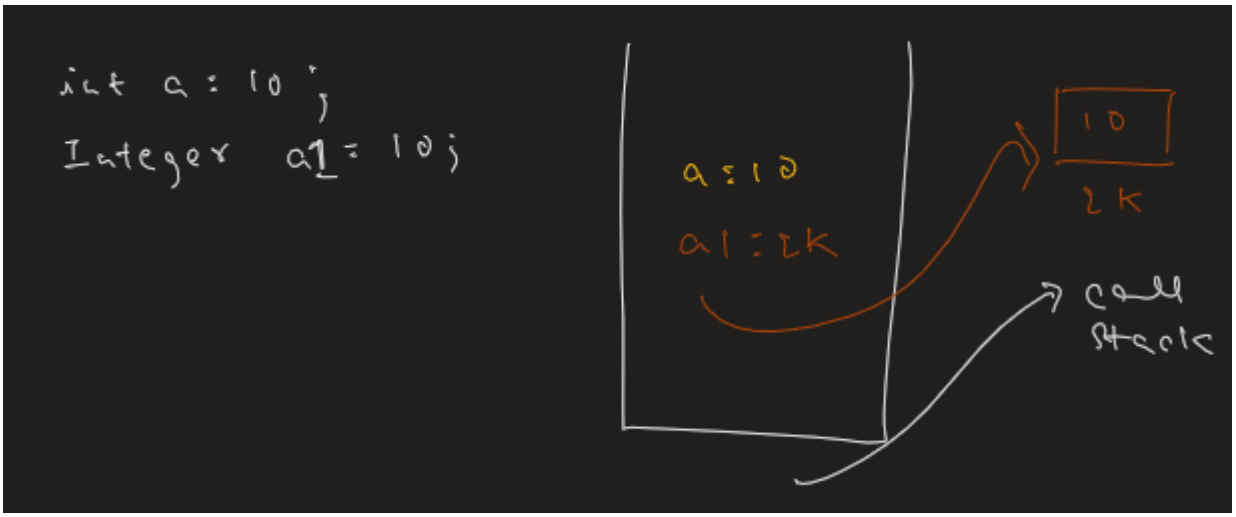
Java is an **object-oriented language**, but its primitive types (`int`, `char`, etc.) are **not objects**. To bridge this gap, Java provides **wrapper classes** for each primitive type. Also primitive data types are not allowed at many places in Java.

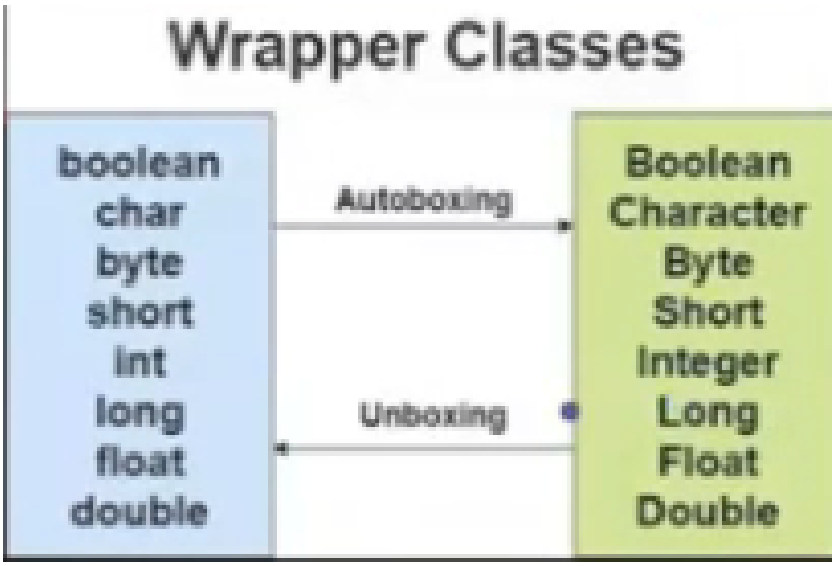
Primitive and wrapper classes are interchangeable.

📦 Primitive Types vs. Wrapper Classes

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`

int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean






```java
public class Main {
	public static void main(String[] args) {
		int a = 10;
		Integer a1 = 10; // auto-boxing
		System.out.println(a);
		System.out.println(a1);
		Long l = 89l;
		long ll = 7890;
		// l=ll;
		a1 = a; // auto-boxing
		ll = l; // un-boxing

		Integer c1 = 102;
```

```
            Integer c2 = 102;
            Integer c3 = 722;
            Integer c4 = 722;
            System.out.println(c1 == c2); // true
            System.out.println(c3 == c4); // false
            System.out.println(c3.equals(c4)); // true

            Character ch1 = 'a';
            Character ch2 = 'a';
            System.out.println(ch1 == ch2); // true

            Boolean b1 = true;
            Boolean b2 = true;
            System.out.println(b1 == b2); // true

            Double d1 = 189.7;
            Double d2 = 189.7;
            System.out.println(d1 == d2);  // false

            System.out.println(a == a1); // content  // true
        }
}
```

🔍 Understanding Wrapper Class Caching and Equality

- `==` **compares object references**, not values (except for primitives).
- `.equals()` **compares values**.
- **Java caches small values** of certain wrapper types to save memory and improve performance.
- Caching only applies when you're using **autoboxing** or `valueOf()`, **not** when using `new`.

📦 Wrapper Classes Overview

Type	Cached Range (when autoboxed)	Notes
`Byte`	**All values** (-128 to 127)	Fully cached
`Short`	-128 to 127	Like Integer
`Integer`	-128 to 127	Most common case
`Long`	-128 to 127	Like Integer
`Character`	`\u0000` to `\u007F` (0 to 127)	ASCII range
`Boolean`	Only `true` and `false`	Always cached
`Float`	❌ Not cached	Always new instances
`Double`	❌ Not cached	Always new instances

```java
Integer a = 100;
Integer b = 100;
System.out.println(a == b); // true (within cache)

Integer c = 200;
Integer d = 200;
System.out.println(c == d); // false (outside cache)

Long e = 127L;
Long f = 127L;
System.out.println(e == f); // true

Long g = 128L;
Long h = 128L;
System.out.println(g == h); // false

Character ch1 = 65;
Character ch2 = 65;
System.out.println(ch1 == ch2); // true (within cache)

Character ch3 = 200;
Character ch4 = 200;
System.out.println(ch3 == ch4); // false

Boolean bool1 = true;
Boolean bool2 = true;
System.out.println(bool1 == bool2); // true

Float fl1 = 10.5f;
Float fl2 = 10.5f;
System.out.println(fl1 == fl2); // false

Double d1 = 10.5;
Double d2 = 10.5;
System.out.println(d1 == d2); // false
```

💡 Tips

1. **Use `.equals()`** when comparing wrapper objects for value equality.
2. Prefer **autoboxing (`Integer i = 10`) or `valueOf()`** instead of `new`, to benefit from caching.
3. If you're dealing with `Float` and `Double`, always use `.equals()` — they are **not cached**.
4. **Primitives (`int`, `long`, etc.)** are compared **by value** with `==` — no issue there.

📚 What is an ArrayList in Java?

An **ArrayList** is a **resizable array** implementation provided by Java's `java.util` package. Unlike regular arrays, which have a fixed size, an ArrayList can **grow or shrink dynamically** as elements are added or removed.

It is a **non-primitive data type**, meaning it is a class and resides in **heap memory**.

🧠 Key Features of ArrayList

- ✅ **Dynamic sizing**: No need to declare a fixed size
- ✅ **Indexed access**: Elements accessed using `get(index)`
- ✅ **Built-in methods**: Add, remove, search, sort, etc.
- ✅ **Type-safe**: Can be generic (e.g., `ArrayList<String>`)

🇮🇳 Real-Life Analogy: Street Food Thali

Imagine you're at a **chaat stall** in Lucknow:

- You start with a basic thali (plate).
- As you taste more items, you keep adding: aloo tikki, papdi chaat, dahi puri…
- You can remove items too—no fixed size!

That's an ArrayList: a flexible thali that adjusts to your appetite.

🇮🇳 Real-Life Analogy: The Wedding Guest List

Imagine you're organizing a big **Indian wedding** in Noida. You start with a guest list:

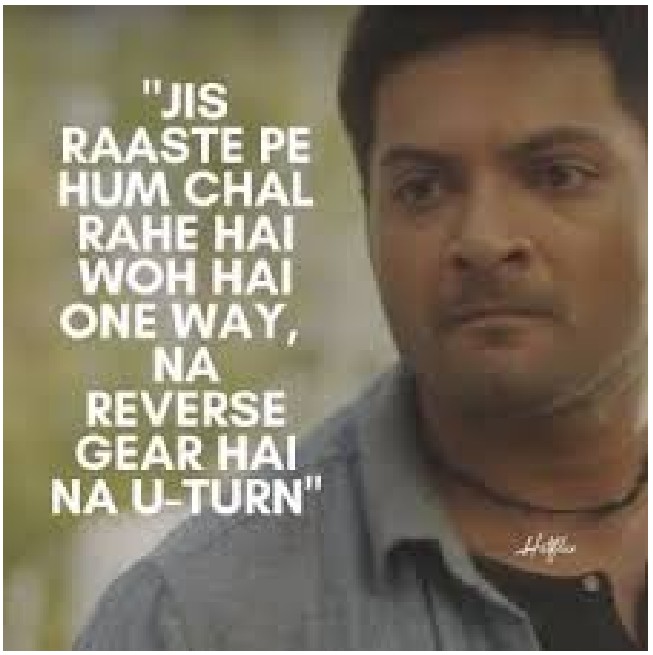
- Initially, you jot down names of close relatives.
- But as the days go by, your parents add more names—friends, neighbors, distant cousins.
- You also remove a few names (maybe someone can't attend).
- You keep checking who's on the list, how many people are coming, and whether someone is already invited.

This evolving guest list is exactly how an **ArrayList** works.

```java
import java.util.ArrayList;
import java.util.Collections;

public class Main {
    public static void main(String[] args) {

        // Create an ArrayList of Integers
        ArrayList<Integer> ll = new ArrayList<>();

        // Print the empty list and its size
        System.out.println(ll);           // []
        System.out.println(ll.size());    // 0

        // Add elements to the list
        ll.add(10);
        ll.add(3);
        ll.add(20);
        ll.add(4);

        // Insert -2 at index 2
        ll.add(2, -2);                    // [10, 3, -2, 20, 4]

        // Print the list and its size
        System.out.println(ll);          // [10, 3, -2, 20, 4]
        System.out.println(ll.size());   // 5

        // Access and print element at index 2
        System.out.println(ll.get(2));   // -2
```

```
        // Remove element at index 1 and print the removed value
        System.out.println(ll.remove(1)); // 3

        // Print the list after removal
        System.out.println(ll);          // [10, -2, 20, 4]

        // Update value at index 1 to 90
        ll.set(1, 90);                   // [10, 90, 20, 4]

        // Sort the list in ascending order
        Collections.sort(ll);            // [4, 10, 20, 90]
        System.out.println(ll);

        // Reverse the list (O(n) time)
        Collections.reverse(ll);         // [90, 20, 10, 4]
        System.out.println(ll);
    }
}
```

🔁 For each loop




```
for (int i = 0; i < ll.size(); i++) {
    System.out.print(ll.get(i) + " ");
}
System.out.println();

int[] arr = new int[5];
for (int x : arr) {
    System.out.print(x + " ");
}
System.out.println();

for (int x : ll) { // un-boxing
    System.out.print(x + " ");
}
```

📦 Difference Between Size and Capacity in ArrayList

🔹 Real-Life Analogy: The Sugar Godown

Imagine you own a **godown** in Dadri that stores boxes of sugar.

- You currently have **40 boxes** stored — that's the **size**.
- The godown can hold up to **5,000 boxes** — that's the **capacity**.

In Java's `ArrayList`, the same logic applies:

- **Size** is the number of elements currently stored.
- **Capacity** is the total space available before resizing is needed.

🧮 ArrayList Initialization and Capacity Behavior

🔹 Default Capacity

When you declare an `ArrayList` without specifying capacity, the default **initial capacity** is **10**.

```
ArrayList<String> list = new ArrayList<>();
```

🔹 Custom Capacity

You can set the initial capacity explicitly:

```
ArrayList<String> list = new ArrayList<>(5000); // Capacity = 5000
```

This does **not** mean the list has 5000 elements—it just reserves space for up to 5000 before resizing.

🔄 How ArrayList Grows Internally

When you exceed the current capacity:

- A **new, larger array** is created.
- All elements from the old array are **copied** into the new one.
- The old array is **discarded** and cleaned up by the **Garbage Collector**.

🔹 Growth Formula:

new capacity = old capacity + (old capacity / 2) + 1
This means the capacity grows by approximately **1.5×** each time.

🔹 Example:

- Initial capacity: 10
- Add 11th element → triggers resize
- New capacity becomes: `10 + 5 + 1 = 16`